6

AJAX

В последние годы стало общепринятым предосудительное отношение к сайтам, которые основаны на применении нестандартных технологий. Одним из самых известных словечек, используемых для описания новых веб-приложений, стало **AJAX-Powered** (управляется технологиями AJAX). Этот термин употребляется для обозначения самых разных вещей, так как охватывает целую группу взаимосвязанных технологий.

Формально слово **AJAX** является аббревиатурой от **Asynchronous JavaScript and XML** (асинхронный JavaScript и XML). В число технологий, включенных в решение AJAX, входят:

- *JavaScript* для организации взаимодействий с пользователем и обработки событий, происходящих в броузерах
- Объект **XMLHttpRequest**, который позволяет выполнять запросы к серверу без прерывания выполнения других задач в броузере
- Файлы в формате *XML* или в других подобных форматах, таких как HTML или JSON, на стороне сервера
- И снова *JavaScript* для интерпретации данных, полученных от сервера, и их отображения на странице

Технологии AJAX были провозглашены средством спасения Всемирной паутины, позволяющим превратить статические веб-страницы в интерактивные веб-приложения. Было создано множество платформ, призванных помочь разработчикам, использующим эти технологии, решить проблему несовместимости реализаций объекта `XMLHttpRequest` в разных броузерах. И библиотека jQuery не исключение.

В этой главе мы попытаемся разобраться, действительно ли AJAX позволяет творить чудеса.

Загрузка данных по требованию

Если отбросить всю эту рекламную шумиху, AJAX – это всего лишь средство загрузки данных с **сервера** в веб-броузер, или **клиент**, не вызывающее видимого обновления страницы. Эти данные могут принимать самые разные формы, и при получении их они могут обрабатываться массой разных способов. Мы убедимся в этом, решая одну и ту же простую задачу разными путями.

Мы создадим страницу, которая будет отображать словарные статьи, сгруппированные по первому символу статьи. Ниже приводится разметка HTML, определяющая область содержимого страницы:

```
<div id="dictionary">
</div>
```

Это не ошибка! Изначально страница не содержит никакой информации. Мы будем использовать различные методы поддержки технологии AJAX, входящие в состав библиотеки jQuery, чтобы заполнить этот элемент <div> словарными статьями.

Нам необходим механизм, запускающий процесс загрузки данных, поэтому мы добавим несколько ссылок, к которым будем подключать **обработчики событий**:

```
<div class="letters">
  <div class="letter" id="letter-a">
    <h3><a href="#">A</a></h3>
  </div>
  <div class="letter" id="letter-b">
    <h3><a href="#">B</a></h3>
  </div>
  <div class="letter" id="letter-c">
    <h3><a href="#">C</a></h3>
  </div>
  <div class="letter" id="letter-d">
    <h3><a href="#">D</a></h3>
  </div>
</div>
```

Примечание

Как обычно, действующая реализация должна предусматривать возможность **прогрессивного улучшения**, чтобы обеспечить работоспособность страницы в броузерах с отключенной поддержкой JavaScript. Здесь, чтобы упростить пример, мы не предусматриваем никаких действий для ссылок, пока с помощью jQuery к ним не будут подключены обработчики событий.

Добавив несколько правил CSS, мы получили страницу, которая выглядит, как показано на рис. 6.1.

Рис. 6.1. *Начальная страница после добавления некоторых правил CSS*

Теперь можно сосредоточиться на наполнении страницы содержимым.

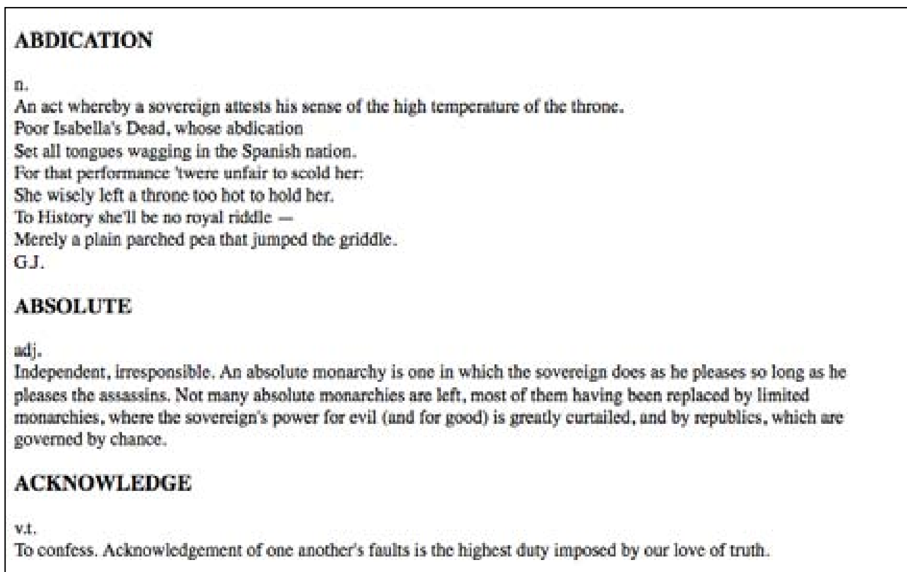
Добавление разметки HTML

Приложения с поддержкой технологии AJAX зачастую просто запрашивают получение фрагментов разметки HTML. Этот прием, который иногда называется **AHAH (Asynchronous HTTP and HTML – асинхронный HTTP и HTML)**, чрезвычайно просто реализуется с помощью jQuery. Во-первых, нам необходима некоторая разметка HTML для добавления в страницу, которую мы поместим в файл с именем `a.html`, находящийся в одном каталоге с нашим документом. Ниже приводится начало этого файла HTML:

```
<div class="entry">
  <h3 class="term">ABDICATION</h3>
  <div class="part">n.</div>
  <div class="definition">
    An act whereby a sovereign attests his sense of the high
                          temperature of the throne.
    <div class="quote">
      <div class="quote-line">Poor Isabella's Dead, whose
                          abdication</div>
      <div class="quote-line">Set all tongues wagging in the
                          Spanish nation.</div>
      <div class="quote-line">For that performance 'twere
                          unfair to scold her:</div>
      <div class="quote-line">She wisely left a throne too
                          hot to hold her.</div>
      <div class="quote-line">To History she'll be no royal
                          riddle &mdash;</div>
      <div class="quote-line">Merely a plain parched pea that
                          jumped the griddle.</div>
      <div class="quote-author">G.J.</div>
    </div>
  </div>
</div>
```

```
<div class="entry">
  <h3 class="term">ABSOLUTE</h3>
  <div class="part">adj.</div>
  <div class="definition">
    Independent, irresponsible. An absolute monarchy is one
    in which the sovereign does as he pleases so long as he
    pleases the assassins. Not many absolute monarchies are
    left, most of them having been replaced by limited
    monarchies, where the sovereign's power for evil (and for
    good) is greatly curtailed, and by republics, which are
    governed by chance.
  </div>
</div>
```

Страница содержит множество статей с такой же структурой разметки HTML. Будучи самостоятельной страницей, она выглядит очень просто, как показано на рис. 6.2.



ABDICATION

n.
An act whereby a sovereign attests his sense of the high temperature of the throne.
Poor Isabella's Dead, whose abdication
Set all tongues wagging in the Spanish nation.
For that performance 'twere unfair to scold her:
She wisely left a throne too hot to hold her.
To History she'll be no royal riddle —
Merely a plain parched pea that jumped the griddle.
G.J.

ABSOLUTE

adj.
Independent, irresponsible. An absolute monarchy is one in which the sovereign does as he pleases so long as he pleases the assassins. Not many absolute monarchies are left, most of them having been replaced by limited monarchies, where the sovereign's power for evil (and for good) is greatly curtailed, and by republics, which are governed by chance.

ACKNOWLEDGE

v.t.
To confess. Acknowledgement of one another's faults is the highest duty imposed by our love of truth.

Рис. 6.2. *Внешний вид страницы со словарными статьями*

Обратите внимание, что файл a.html не является настоящим документом HTML – в нем отсутствуют теги <html>, <head> и <body>, которые являются обязательными. Обычно мы называем такие файлы **отрывками** или **фрагментами**, их основное назначение – хранить разметку HTML, которая будет вставляться в другой документ HTML, которым мы сейчас и займемся:

```
$(document).ready(function() {
  $('#letter-a a').click(function() {
    $('#dictionary').load('a.html');
    return false;
  });
});
```

Метод .load() выполняет всю тяжелую работу! С помощью обычного селектора jQuery мы указали место, куда будет вставляться отрывок, и затем передали методу адрес URL загружаемого файла в виде аргумента. Теперь, когда на ссылке будет выполнен первый щелчок, файл будет загружен, а его содержимое вставлено в элемент <div id="dictionary">. Броузер отобразит новую разметку HTML после ее добавления в документ, как показано на рис. 6.3.

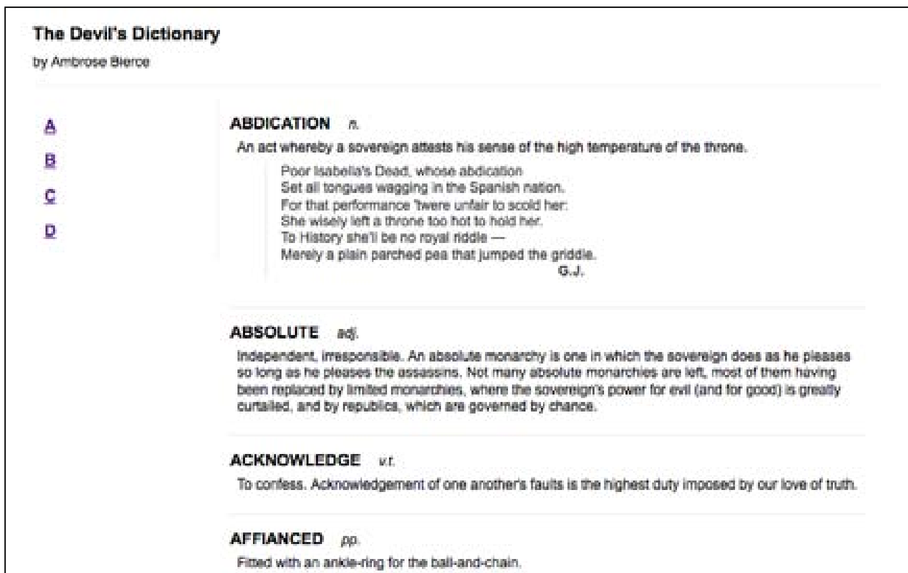


Рис. 6.3. *Вид страницы после добавления нового фрагмента разметки HTML*

Обратите внимание, что прежде простая разметка HTML приобрела оформление. Это обусловлено правилами CSS в основном документе – когда в документ вставляется новый фрагмент HTML, к его тегам применяются правила оформления, существующие в основном документе.

При опробовании этого примера вы наверняка увидите, что словарные определения будут появляться практически мгновенно, сразу же после щелчка на ссылке. В этом заключается опасность работы с локальными приложениями – в такой ситуации очень сложно учесть задержки, возникающие при передаче документов через сеть. Предположим,

что мы добавили диалог, сообщающий о завершении загрузки определений:

```
$(document).ready(function() {
  $('#letter-a a').click(function() {
    $('#dictionary').load('a.html');
    alert('Loaded!');
    return false;
  });
});
```

Из структуры этого программного кода можно было бы предположить, что диалог будет отображаться только после того, как загрузка данных будет завершена. Программный код JavaScript обычно выполняется **синхронно**, действия происходят в строгой последовательности друг за другом.

Однако если опробовать этот программный код на действующем веб-сервере, из-за задержек, возникающих в сети, диалог наверняка появится прежде, чем загрузка будет завершена. Это происходит потому, что все запросы AJAX по умолчанию выполняются **асинхронно**. В противном случае нам пришлось бы назвать эту технологию SJAX, что едва ли вызвало бы точно такую же шумиху! В асинхронном режиме загрузки, сразу после того как будет отправлен запрос HTTP на получение фрагмента разметки HTML, сценарий немедленно продолжит свою работу. Спустя некоторое время броузер примет ответ от сервера и обработает его. В большинстве случаев именно такой порядок работы является желаемым – было бы недружелюбным по отношению к пользователю блокировать весь броузер, ожидая получения данных.

Для случаев, когда действие должно быть отложено до завершения загрузки данных, библиотека jQuery предоставляет возможность определить **функцию обратного вызова**, реализующую эти действия. Пример такой функции будет представлен ниже.

Работа с объектами JavaScript

Запрашивать уже готовый фрагмент разметки HTML очень удобно, но иногда бывает необходимо, чтобы сценарий выполнял предварительную обработку данных, прежде чем отобразить их. В этом случае нам необходимо организовать получение данных в виде структуры, обход которой можно будет выполнить с помощью JavaScript.

Используя селекторы jQuery, мы могли бы реализовать обход получаемой разметки HTML и выполнить требуемые операции, но для этого она сначала должна быть вставлена в документ. Для обработки данных в формате, который ближе к языку JavaScript, может даже потребоваться меньше программного кода.

Извлечение объектов JavaScript

Зачастую нам встречаются **объекты JavaScript**, которые являются всего лишь множеством **пар ключ-значение** и могут быть определены с помощью фигурных скобок (**{}**). С другой стороны, массивы JavaScript могут определяться в ходе выполнения с помощью квадратных скобок (**[]**). Комбинируя эти две концепции, мы легко можем выразить самые сложные структуры данных.

Для обозначения этого простого синтаксиса Дугласом Крокфордом (Douglas Crockford) был придуман термин **JSON (JavaScript Object Notation – форма записи объектов JavaScript).** Эта форма записи может рассматриваться как более компактная альтернатива более расточительному формату XML:

```
{
  "key": "value",
  "key 2": [
    "array",
    "of",
    "items"
  ]
}
```

Примечание

Дополнительную информацию о потенциальных преимуществах формата JSON и его реализациях во многих языках программирования можно найти по адресу: *http://json.org/*.

Представить свои данные в этом формате можно самыми разными способами. Некоторые словарные статьи в формате JSON мы поместим в файл с именем `b.json`, начало которого приводится ниже:

```
[
  {
    "term": "BACCHUS",
    "part": "n.",
    "definition": "A convenient deity invented by the...",
    "quote": [
      "Is public worship, then, a sin,",
      "That for devotions paid to Bacchus",
      "The lictors dare to run us in,",
      "And resolutely thump and whack us?"
    ],
    "author": "Jorace"
  },
  {
    "term": "BACKBITE",
    "part": "v.t.",
    "definition": "To speak of a man as you find him when..."
  },
```

```
{
  "term": "BEARD",
  "part": "n.",
  "definition": "The hair that is commonly cut off by..."
},
```

Для извлечения этих данных мы будем использовать метод `$.getJSON()`, который извлекает файл и обрабатывает его, возвращая вызывающему программному коду сформированный объект JavaScript.

Глобальные функции jQuery

До этого момента мы использовали только методы объекта jQuery, создаваемого фабричной функцией `$()`. Селекторы позволяли нам определять множество требуемых узлов дерева DOM, а методы давали возможность воздействовать на них тем или иным способом. Однако функция `$.getJSON()` – это совсем другое дело. Здесь нет элемента DOM, к которому логично было применить эту функцию – возвращаемый объект должен передаваться сценарию, а не добавляться в страницу. По этой причине функция `getJSON()` определена как метод **глобального объекта jQuery** (единственный объект с именем `jQuery`, или `$`, определяемый библиотекой jQuery), а не как метод конкретного **экземпляра объекта jQuery** (объекты, которые создаются с помощью функции `$()`).

Если бы в JavaScript имелись классы, похожие на классы в других объектно-ориентированных языках программирования, мы бы назвали `$.getJSON()` **методом класса**. В дальнейшем обсуждении методы такого типа мы будем назвать **глобальными функциями**. В действительности – это функции, использующие **пространство имен** jQuery, чтобы избежать конфликтов с именами других функций.

При использовании этой функции ей передается имя файла, так же как и прежде:

```
$(document).ready(function() {
  $('#letter-b a').click(function() {
    $.getJSON('b.json');
    return false;
  });
});
```

Этот программный код не производит никаких видимых эффектов при щелчке мышью на ссылке. Функция загружает файл, но мы еще не сказали JavaScript, что делать с полученными данными. Для этого нам потребуется задействовать **функцию обратного вызова**.

Функция `$.getJSON()` принимает во втором аргументе функцию, которая будет вызвана по завершении загрузки. Как уже упоминалось выше, вызовы AJAX выполняются **асинхронно**, а функция обратного вызова является способом отложить выполнение операций до момента, пока не будут получены данные. Функция обратного вызова также принима-

ет аргумент, в котором ей передаются полученные данные. Поэтому мы можем написать такой программный код:

```
$(document).ready(function() {
  $('#letter-b a').click(function() {
    $.getJSON('b.json', function(data) {
    });
    return false;
  });
});
```

Здесь в качестве функции обратного вызова мы использовали **анонимную функцию**, что часто используется для сокращения объема программного кода при работе с библиотекой jQuery. С тем же успехом в качестве функции обратного вызова можно было бы использовать именованную функцию.

Внутри этой функции мы можем использовать переменную `data` для обхода структуры с данными. Нам потребуется выполнить итерации по элементам массива и для каждого из них создать разметку HTML. Это можно было бы реализовать с помощью стандартного цикла `for`, но вместо этого мы познакомимся с другой глобальной функцией jQuery – `$.each()`. В главе 5 мы уже встречались с родственным ей методом `.each()`. Но в отличие от него, эта функция работает не с объектом jQuery, а принимает массив или отображение в первом аргументе и функцию обратного вызова – во втором. На каждой итерации производится обращение к функции обратного вызова, которой в виде двух аргументов передаются текущий **порядковый номер итерации** и текущий элемент массива или отображения.

```
$(document).ready(function() {
  $('#letter-b a').click(function() {
    $.getJSON('b.json', function(data) {
      $('#dictionary').empty();
      $.each(data, function(entryIndex, entry) {
        var html = '<div class="entry">';
        html += '<h3 class="term">' + entry['term'] + '</h3>';
        html += '<div class="part">' + entry['part'] + '</div>';
        html += '<div class="definition">';
        html += entry['definition'];
        html += '</div>';
        html += '</div>';
        $('#dictionary').append(html);
      });
    });
    return false;
  });
});
```

Перед началом цикла выполняется очистка содержимого элемента `<div id="dictionary">`, чтобы мы могли наполнить его вновь созданной размет-

кой HTML. Затем с помощью функции `$.each()` поочередно извлекается каждый элемент массива и на основе его содержимого создается структура HTML. В заключение полученная разметка HTML вставляется в дерево DOM посредством добавления ее в элемент `<div>`.

Примечание

Такой подход предполагает, что полученные данные можно без опаски вставлять в разметку HTML, то есть, они не содержат специальных символов, таких как <.

Все, что осталось сделать, – это предусмотреть обработку статей с цитатами, которая выполняется с помощью другого цикла `$.each()`:

```
$(document).ready(function() {
  $('#letter-b a').click(function() {
    $.getJSON('b.json', function(data) {
      $('#dictionary').empty();
      $.each(data, function(entryIndex, entry) {
        var html = '<div class="entry">';
        html += '<h3 class="term">' + entry['term'] + '</h3>';
        html += '<div class="part">' + entry['part'] + '</div>';
        html += '<div class="definition">';
        html += entry['definition'];
        if (entry['quote']) {
          html += '<div class="quote">';
          $.each(entry['quote'], function(lineIndex, line) {
            html += '<div class="quote-line">' + line + '</div>';
          });
          if (entry['author']) {
            html += '<div class="quote-author">' + entry['author'] + '</div>';
          }
          html += '</div>';
        }
        html += '</div>';
        html += '</div>';
        $('#dictionary').append(html);
      });
    });
    return false;
  });
});
```

Добавив этот программный код, можно попробовать щелкнуть на ссылке B и увидеть результаты, как показано на рис. 6.4.

Примечание

Формат JSON обеспечивает компактность представления данных, но он не прощает ошибок. Каждая скобка, кавычка и запятая должны находиться на своем месте, иначе файл не будет загружен. В большинстве броузеров вы даже не получите сообщение об ошибке – сценарий просто будет терпеть неудачу.

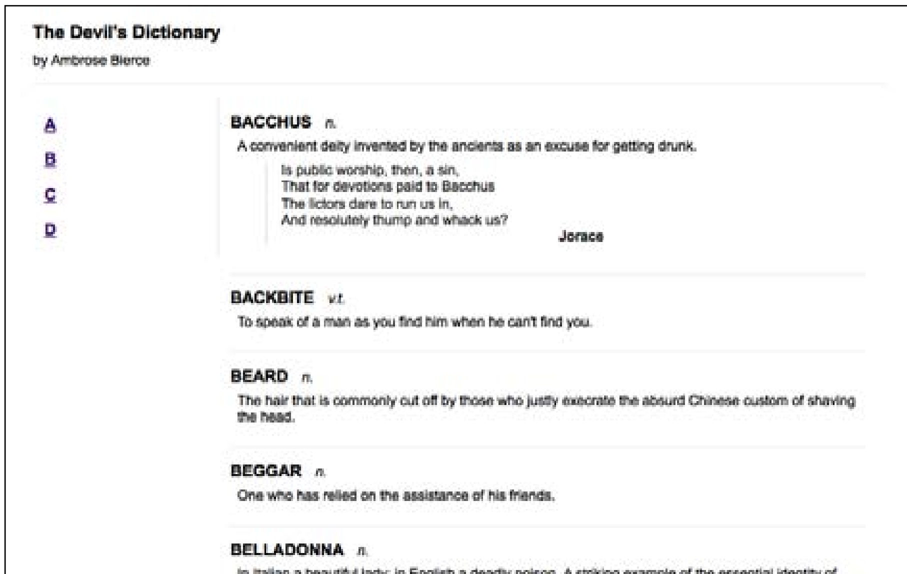


Рис. 6.4. *Вид страницы после загрузки данных в формате JSON*

Запуск сценария

Иногда требуется, чтобы не весь программный код JavaScript загружался вместе со страницей. Мы можем не знать, какие сценарии потребуются, пока пользователь не выполнит какие-либо действия. В такой ситуации можно «на лету» генерировать теги `<script>`, по мере необходимости, однако существует более элегантный способ внедрения дополнительного программного кода, который заключается в том, чтобы с помощью jQuery напрямую загружать файл `.js`.

Загрузка сценария выполняется так же просто, как и загрузка фрагмента HTML. Для этого можно воспользоваться глобальной функцией `$.getScript()`, которая, как и родственные ей функции, принимает адрес URL файла сценария:

```
$(document).ready(function() {
  $('#letter-c a').click(function() {
    $.getScript('c.js');
    return false;
  });
});
```

В нашем последнем примере нам требовалось обработать полученные данные, чтобы можно было с пользой использовать загруженный файл. В случае же с файлом сценария его обработка выполняется автоматически – сценарий просто запускается.

Сценарии, полученные таким способом, выполняются в **глобальном контексте** текущей страницы. Это означает, что они имеют доступ ко всем глобальным функциям и переменным, включая и библиотеку jQuery. Благодаря этому мы можем сымитировать работу примера, использующего данные в формате JSON, для подготовки и добавления в страницу разметки HTML во время выполнения сценария и поместить этот программный код в отдельный файл c.js:

```
var entries = [
{
  "term": "CALAMITY",
  "part": "n.",
  "definition": "A more than commonly plain and..."
},
{
  "term": "CANNIBAL",
  "part": "n.",
  "definition": "A gastronome of the old school who..."
},
{
  "term": "CHILDHOOD",
  "part": "n.",
  "definition": "The period of human life intermediate..."
},
{
  "term": "CLARIONET",
  "part": "n.",
  "definition": "An instrument of torture operated by..."
},
{
  "term": "COMFORT",
  "part": "n.",
  "definition": "A state of mind produced by..."
},
{
  "term": "CORSAIR",
  "part": "n.",
  "definition": "A politician of the seas."
}
];

var html = '';

$.each(entries, function() {
  html += '<div class="entry">';
  html += '<h3 class="term">' + this['term'] + '</h3>';
  html += '<div class="part">' + this['part'] + '</div>';
  html += '<div class="definition">' + this['definition'] + '</div>';
  html += '</div>';
});
$('#dictionary').html(html);
```

Теперь щелчок на ссылке `C` будет давать желаемый результат, как показано на рис. 6.5.

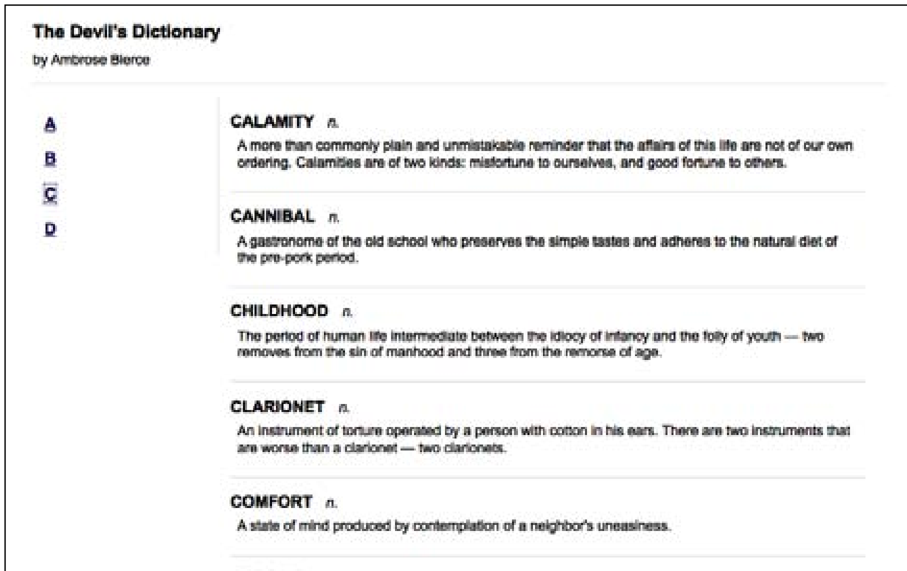


Рис. 6.5. *Вид страницы после загрузки данных с дополнительным сценарием*

Загрузка документа XML

Название XML является частью аббревиатуры AJAX, но мы еще не рассматривали загрузку данных в формате XML. Эта возможность реализуется просто и очень похожа на методику работы с форматом JSON. Для начала нам потребуется XML-файл `d.xml`, содержащий некоторые данные, которые требуется отобразить:

```
<?xml version="1.0" encoding="UTF-8"?>
<entries>
  <entry term="DEFAME" part="v.t.">
    <definition>
      To lie about another. To tell the truth about another.
    </definition>
  </entry>
  <entry term="DEFENCELESS" part="adj.">
    <definition>
      Unable to attack.
    </definition>
  </entry>
  <entry term="DELUSION" part="n.">
    <definition>
      The father of a most respectable family, comprising
      Enthusiasm, Affection, Self-denial, Faith, Hope,
```

```
      Charity and many other goodly sons and daughters.
    </definition>
    <quote author="Mumfrey Mappel">
      <line>All hail, Delusion! Were it not for thee</line>
      <line>The world turned topsy-turvy we should see;</line>
      <line>For Vice, respectable with cleanly fancies,</line>
      <line>Would fly abandoned Virtue's gross advances.</line>
    </quote>
  </entry>
  <entry term="DIE" part="n.">
    <definition>
      The singular of "dice." We seldom hear the word,
      because there is a prohibitory proverb, "Never say
      die." At long intervals, however, some one says: "The
      die is cast," which is not true, for it is cut. The
      word is found in an immortal couplet by that eminent
      poet and domestic economist, Senator Depew:
    </definition>
    <quote>
      <line>A cube of cheese no larger than a die</line>
      <line>May bait the trap to catch a nibbling mie.</line>
    </quote>
  </entry>
</entries>
```

Конечно, эти данные можно выразить множеством способов, и некоторые из них более близко имитируют структуру, выбранную для форматов HTML и JSON, использовавшихся выше. Однако здесь мы постарались продемонстрировать некоторые особенности XML, которые обеспечивают более высокую удобочитаемость этого формата для человека, например использование **атрибутов** для определения терминов и разделов вместо отдельных **тегов**.

Начало функции уже знакомо нам:

```
$(document).ready(function() {
  $('#letter-d a').click(function() {
    $.get('d.xml', function(data) {

    });
    return false;
  });
});
```

На этот раз основную работу будет выполнять функция `$.get()`. Вообще говоря, эта функция просто извлекает файл из заданного адреса URL и передает функции обратного вызова простой текст. Однако если из ответа известно, что он имеет формат XML, благодаря тому, что сервер вместе с ответом возвращает его **тип MIME**, функция обратного вызова получит дерево DOM XML.

К счастью, как мы уже видели, библиотека jQuery обладает мощными средствами обхода дерева DOM. При обработке документов XML мы можем использовать обычные методы .find(), .filter() и другие методы обхода дерева, как если бы это был документ HTML:

```
$(document).ready(function() {
  $('#letter-d a').click(function() {
    $.get('d.xml', function(data) {
      $('#dictionary').empty();
      $(data).find('entry').each(function() {
        var $entry = $(this);
        var html = '<div class="entry">';
        html += '<h3 class="term">' + $entry.attr('term') + '</h3>';
        html += '<div class="part">' + $entry.attr('part') + '</div>';
        html += '<div class="definition">';
        html += $entry.find('definition').text();
        var $quote = $entry.find('quote');
        if ($quote.length) {
          html += '<div class="quote">';
          $quote.find('line').each(function() {
            html += '<div class="quote-line">' + $(this).text() + '</div>';
          });
          if ($quote.attr('author')) {
            html += '<div class="quote-author">'
              + $quote.attr('author') + '</div>';
          }
          html += '</div>';
        }
        html += '</div>';
        html += '</div>';
        $('#dictionary').append($(html));
      });
    });
    return false;
  });
});
```

Этот программный код воспроизводит требуемый эффект, когда выполняется щелчок на ссылке D, как показано на рис. 6.6.

Этот новый способ использования уже знакомых нам методов обхода дерева DOM подчеркивает гибкость поддержки селекторов CSS в библиотеке jQuery. Синтаксис стилей CSS обычно используется для улучшения оформления страниц HTML, поэтому для определения селекторов в стандартных файлах .css используются имена тегов HTML, такие как div или body, чтобы указать местоположение искомого содержимого. Однако при работе с библиотекой jQuery наравне со стандартными именами тегов HTML допускается использовать произвольные имена тегов XML, такие как entry и definition.

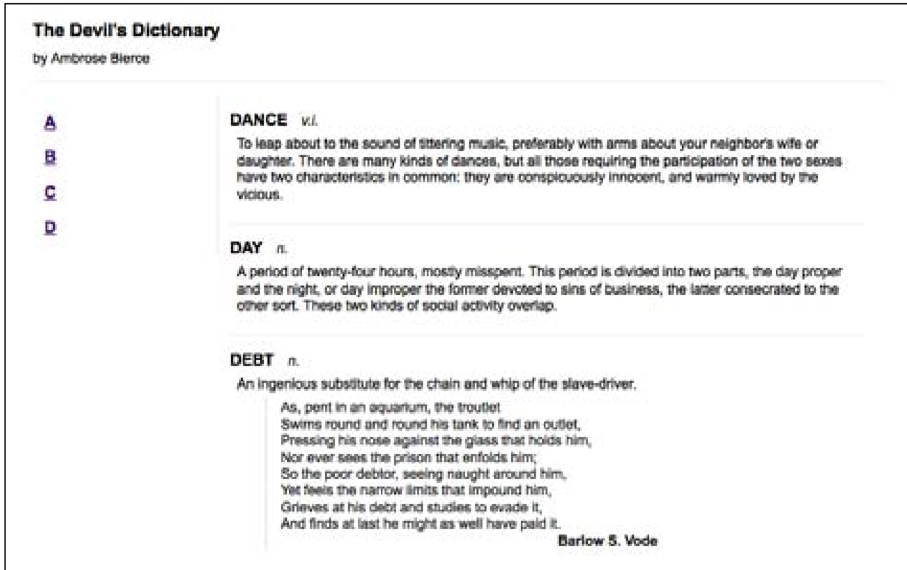


Рис. 6.6. *Вид страницы после загрузки данных в формате XML*

Улучшенный механизм селекторов, реализованный в jQuery, упрощает поиск элементов документа XML даже в самых сложных ситуациях. Например, предположим, что нам требуется отобразить только те статьи, в которых имеются цитаты, для которых в свою очередь указаны авторы. Для этого мы можем организовать выборку только тех статей, которые имеют вложенные элементы `<quote>`, заменив селектор `entry` на `entry:has(quote)`. Затем можно добавить дополнительное ограничение и отобрать только те статьи, в которых элемент `<quote>` имеет атрибут `author`, указав селектор `entry:has(quote[author])`. Теперь строка с начальным селектором будет выглядеть так:

```
$(data).find('entry:has(quote[author])').each(function() {
```

Этот новый селектор соответствующим образом ограничивает перечень возвращаемых словарных статей, как показано на рис. 6.7.

Выбор формата данных

Мы рассмотрели четыре формата для внешних данных, каждый из которых обрабатывается функциями поддержки технологии AJAX из библиотеки jQuery. Мы также убедились, что все четыре формата позволяют решить поставленную задачу загрузки информации в существующую страницу по требованию пользователя, но не раньше. Но как тогда решить, какой из них использовать в своих приложениях?

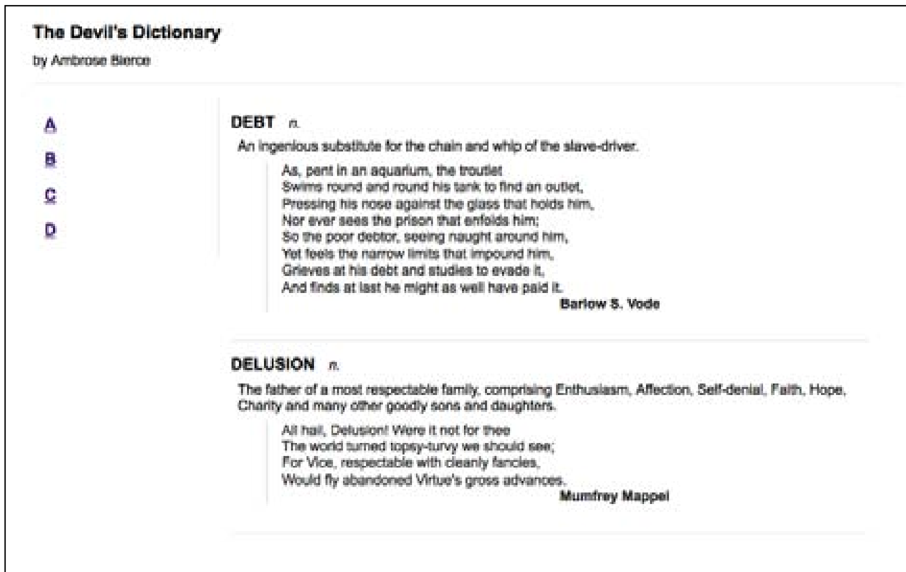


Рис. 6.7. *Вид страницы с ограниченным набором словарных статей*

Фрагменты HTML требуют очень небольшого объема программного кода для его обработки. Внешние данные могут загружаться и вставляться в страницу одним простым методом, который даже не требует функции обратного вызова. Для решения простой задачи добавления новой разметки HTML в существующую страницу нет необходимости обращаться к методам обхода элементов дерева DOM. С другой стороны, данные необязательно могут быть структурированы так, чтобы их можно было повторно использовать в других приложениях. Внешний файл слишком тесно связан с контейнером, куда он вставляется.

Файлы JSON структурированы и легко могут использоваться повторно. Они компактны, и легко читаются. Чтобы извлечь информацию из такого файла и отобразить ее на странице, необходимо выполнить обход структуры данных, но это можно реализовать с помощью стандартных средств JavaScript. Разбор файлов может производиться единственным вызовом функции `eval()` языка JavaScript, поэтому чтение файла в формате JSON выполняется чрезвычайно быстро. Однако любое обращение к функции `eval()` представляет собой угрозу безопасности. Ошибки в файле JSON могут вызывать завершение работы сценария без вывода сообщений об ошибках или побочные эффекты, поэтому данные должны подготавливаться с особой тщательностью.

Файлы JavaScript обеспечивают непревзойденную гибкость, но в действительности они не являются механизмом хранения данных. Эти файлы тесно связаны с языком программирования, поэтому они не могут использоваться для передачи той же самой информации системам, отличным от JavaScript. Однако использование файлов JavaScript

позволяет вынести редко используемые операции во внешние файлы, уменьшить объем программного кода и загружать требуемую реализацию только по мере необходимости.

Документы XML обеспечивают наилучшую переносимость. Так как XML стал *общепринятым языком* обмена данными с веб-службами, предоставление данных в этом формате обеспечивает возможность повторного их использования в разных местах. Примерами могут служить такие сайты, как Flickr (*http://flickr.com/*), del.icio.us (*http://del.icio.us/*) и Upcoming (*http://upcoming.org/*), которые экспортируют данные в формате XML, что позволяет использовать их во многих **гибридных приложениях**. Однако формат XML является довольно расточительным, а скорость работы с ним может быть ниже, чем с другими форматами.

Учитывая все эти характеристики, обычно проще организовать передачу данных в виде фрагментов HTML при условии, что они не будут использоваться другими приложениями. В случаях, когда данные могут использоваться и другими приложениями, часто хорошим выбором является формат JSON благодаря высокой скорости обработки и небольшому размеру. Когда об удаленных приложениях ничего не известно, формат XML даст наибольшую гарантию функциональной совместимости с ними.

Но прежде чем делать какой-либо выбор, следует проверить, доступны ли данные. Если они уже доступны, есть вероятность, что они поставляются в одном из описанных форматов, и тогда может оказаться так, что решение уже будет принято за нас.

Передача данных на сервер

До настоящего момента все наше внимание в примерах было сконцентрировано на получении **статических** файлов с данными со стороны сервера. Однако технология AJAX проявляет всю полноту своих возможностей, только когда сервер может **динамически** формировать данные, исходя из информации, получаемой от броузера. Библиотека jQuery помогает в решении и этой задачи – все методы, рассматривавшиеся до сих пор, могут отправлять данные на сервер, превращая канал связи в улицу с двусторонним движением.

Примечание

Поскольку для демонстрации этих приемов необходима возможность взаимодействия с веб-сервером, мы впервые будем использовать программный код, выполняющийся на сервере. Данные примеры написаны на языке сценариев **PHP**, свободно доступном и получившем широкое распространение. Мы не будем рассматривать вопросы, связанные с настройкой веб-сервера и PHP; необходимую помощь можно получить на веб-сайтах проектов Apache (*http://apache.org/*) и PHP (*http://php.net/*) или в компании, которая предоставляет услуги хостинга для вашего веб-сервера.

Выполнение запроса GET

Чтобы продемонстрировать порядок взаимодействия между клиентом и сервером, мы напишем сценарий, который будет возвращать по одной словарной статье на каждый запрос. Выбор статьи зависит от параметров, передаваемых броузером. Наш сценарий будет хранить свои данные во внутренней структуре, такой как показано ниже:

```
<?php
$entries = array(
  'EAVESDROP' => array(
    'part' => 'v.i.',
    'definition' => 'Secretly to overhear a catalogue of the
      crimes and vices of another or yourself.',
    'quote' => array(
      'A lady with one of her ears applied',
      'To an open keyhole heard, inside,',
      'Two female gossips in converse free &mdash;',
      'The subject engaging them was she.',
      '"I think," said one, "and my husband thinks',
      'That she\'s a prying, inquisitive minx!"',
      'As soon as no more of it she could hear',
      'The lady, indignant, removed her ear.',
      '"I will not stay," she said, with a pout,',
      '"To hear my character lied about!"',
    ),
    'author' => 'Gopete Sherany',
  ),
  'EDIBLE' => array(
    'part' => 'adj.',
    'definition' => 'Good to eat, and wholesome to digest, as
      a worm to a toad, a toad to a snake, a snake to a pig,
      a pig to a man, and a man to a worm.',
  ),
  'EDUCATION' => array(
    'part' => 'n.',
    'definition' => 'That which discloses to the wise and
      disguises from the foolish their lack of
      understanding.',
  ),
);
?>
```

Окончательная версия этого примера может хранить информацию в базе данных и загружать ее по мере необходимости. Так как в нашем случае данные являются частью сценария, программный код, извлекающий их, выглядит очень просто. Сценарий проверяет полученные данные и конструирует фрагмент HTML для отображения:

```
<?php
$term = strtoupper($_REQUEST['term']);
```

```
  if (isset($entries[$term])) {
    $entry = $entries[$term];

    $html = '<div class="entry">';
    $html .= '<h3 class="term">';
    $html .= $term;
    $html .= '</h3>';
    $html .= '<div class="part">';
    $html .= $entry['part'];
    $html .= '</div>';
    $html .= '<div class="definition">';
    $html .= $entry['definition'];
    if (isset($entry['quote'])) {
      $html .= '<div class="quote">';
      foreach ($entry['quote'] as $line) {
        $html .= '<div class="quote-line">'. $line .'</div>';
      }
      if (isset($entry['author'])) {
        $html .= '<div class="quote-author">'. $entry['author'] .'</div>';
      }
      $html .= '</div>';
    }
    $html .= '</div>';

    $html .= '</div>';
    print($html);
  }
  ?>
```

Теперь, если послать запрос этому сценарию, который мы назвали e.php, он вернет фрагмент HTML, соответствующий термину, отправленному на сервер в виде параметра запроса **GET**. Например, при попытке обратиться к сценарию, как e.php?term=eavesdrop, мы получим обратно фрагмент, как показано на рис. 6.8.

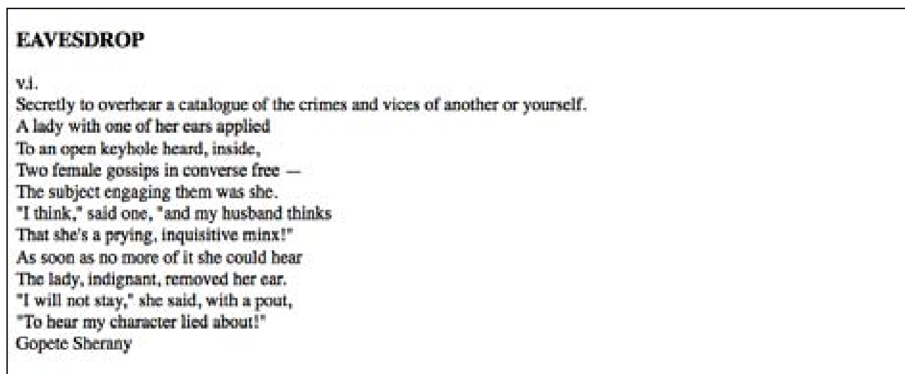


Рис. 6.8. *Фрагмент HTML, соответствующий единственному термину*